ulm university universität
uulm

Institut für Angewandte Informationsverarbeitung
Dr. Andreas F. Borchert

4. November 2012
Blatt 4

Objektorientierte Programmierung mit C++ (WS 2014/2015)

Abgabe bis zum 11. November 2012, 14:00 Uhr

Lernziele:

- Entwickeln einfacher Klassendeklarationen
- Umgang mit Referenzen

Aufgabe 5: Tunnel-Labyrinth

Gegeben sei ein Tunnel-Labyrinth aus n Räumen mit Nummern 0 bis $n-1$. Von jedem der Räume gehen m Tunnel zu anderen Räumen, wobei jede Tunnelverbindung in beiden Richtungen genutzt werden kann. Ein solches Tunnelsystem kann mit einer Eingabedatei spezifiziert werden, in der zuerst n und m genannt und dann für jeden Raum die Tunnelverbindungen spezifiziert werden. Beispiel mit $n = 8$ und $m = 3$:

```
8 3
2 3 7
1 4 8
1 5 6
2 7 8
3 6 7
3 5 8
1 4 5
2 4 6
```

Hierbei beschreibt jede Zeile nach der Anfangszeile einen Raum. Im Beispiel gehen vom Raum 0 Verbindungen zu den Räumen 2, 3 und 7. Wie sich feststellen lässt, sind auch die umgekehrten Richtungen jeweils konsistent angegeben.

Zu entwickeln sind die Klassen *Room* und *Maze* und ein Testprogramm, das ein Labyrinth in der beschriebenen Form einliest und dann ein interaktives Navigieren ermöglicht.

So könnte eine beispielhafte Sitzung aussehen:

```
thales$ Walk
You are in room 0. Tunnels lead to 3, 4, and 7.
Where next? 4
You are in room 4. Tunnels lead to 0, 3, and 5.
Where next? 5
You are in room 5. Tunnels lead to 1, 2, and 4.
Where next? 1
You are in room 1. Tunnels lead to 5, 6, and 7.
Where next? 7
You are in room 7. Tunnels lead to 0, 1, and 6.
Where next? q
Bye!
thales$
```

Hinweise: Als Datenstruktur für die Tunnelverbindungen und die Räume bietet sich wiederum die Container-Klasse *vector* aus der STL an. Hierbei ist es hilfreich, wenn sich die Elementtypen (also **unsigned int** für eine Raumnummer und *Room* für einen Room) ohne Parameter konstruieren lassen. Konkret sollte also *Room* einen parameterlosen Konstruktor haben und die Methoden, die notwendig sind, um den Raum danach zu initialisieren.

Sowohl bei *Room* als auch *Maze* wäre es hilfreich, passende Einlese-Operationen zu haben, mit deren Hilfe das Labyrinth aufgebaut werden kann. Eine Lese-Methode könnte dann etwa so aussehen:

```
bool read(std::istream& in);
   /* PRE: dimension is already known */
```

Der Datentyp *std*::*istream* ist ein beliebiger Datenstrom, von dem gelesen werden kann und der **bool**-Return-Wert spezifiziert, ob das Einlesen erfolgreich war oder nicht. (Sie dürfen alternativ auch die Einlese-Operatoren überdefinieren, aber im Rahmen der Vorlesung beschäftigen wir uns erst später damit.)

Die Klasse *Maze* kann auf ein einzelnes *Room*-Objekt Lesezugriffe einräumen. Das geht mit Hilfe einer Referenz:

```
const Room& get_room(unsigned int room) const;
```

Anderswo können Sie dann eine Referenzvariable zum Alias des *i*-ten Raumes machen:

```
const Room& room{maze.get_room(i)};
```

Es ist dabei entscheidend, dass diese Referenzvariable auch wieder **const** deklariert ist. Auf diese Weise stellt *Maze* die einzelnen Räume nur lesenderweise zur Verfügung und unterbindet damit die Möglichkeit, das Labyrinth nachträglich zu verändern.

Das Einlesen aus einer Datei geht mit Hilfe eines *fstream*-Objekts:

```
#include <fstream>
// ...
int main() {
   std::fstream in{"maze.txt"};
   if (!in) {
      std::cerr << "Sorry, no maze.txt!" << std::endl;
      return 1;
   }
   // ...
}
```

Sie können Ihre Lösung wieder mit *submit* einreichen:

```
thales$ submit cpp 5 Room.hpp Room.cpp Maze.hpp Maze.cpp Walk.cpp
```

Aufgabe 6: Durch die Vereinigten Staaten

Wenn das Tunnelsystem zu abstrakt erscheint, dann gibt es auch eine alternative Aufgabe, bei der jeder „Raum“ eine variable Zahl von „Tunnel“ bzw. Nachbarn hat. Wenn dann noch Raumnummern durch geographische Begriffe ersetzt werden, kann daraus ein Geographie-Spiel werden.

Gegeben sei etwa folgende Spezifikation, hier nur auszugsweise angeführt:[1]

```
49
AL 4 8 9 23 40
AR 6 16 22 23 34 40 41
AZ 4 3 30 31 42
CA 3 2 31 35
CO 6 14 27 30 34 42 48
CT 3 17 32 37
DC 2 18 43
DE 3 18 29 36
FL 2 0 9
...
WY 6 4 11 24 27 39 42
```

Ganz am Anfang steht die Zahl der Lokationen (hier: US-Bundesstaaten einschließlich Washington D. C., die eine zusammenhängende Fläche bilden), gefolgt von genau dieser Zahl von Einträgen. Jeder Eintrag nennt die Lokation (hier jeweils die Zweibuchstaben-Kürzel der US-Bundesstaaten), die Zahl der Nachbarn und dann die zugehörigen Indizes der aufgelisteten Lokationen (beginnend ab 0).

Eine Wanderung von New York (NY) nach Kalifornien (CA) könnte dann so aussehen:

[1] Diese Spezifikation wurde erzeugt mit Hilfe dieser Daten, die von Donald E. Knuth für das Buch *The Stanford GraphBase* veröffentlicht wurden.

```
clonmel$ Walk
You are currently in NY
You can continue your journey to CT MA NJ PA VT
Next: PA
You are currently in PA
You can continue your journey to DE MD NJ NY OH WV
Next: OH
You are currently in OH
You can continue your journey to IN KY MI PA WV
Next: IN
You are currently in IN
You can continue your journey to IL KY MI OH
Next: IL
You are currently in IL
You can continue your journey to IA IN KY MO WI
Next: MO
You are currently in MO
You can continue your journey to AR IA IL KS KY NE OK TN
Next: OK
You are currently in OK
You can continue your journey to AR CO KS MO NM TX
Next: TX
You are currently in TX
You can continue your journey to AR LA NM OK
Next: NM
You are currently in NM
You can continue your journey to AZ CO OK TX
Next: AZ
You are currently in AZ
You can continue your journey to CA NM NV UT
Next: CA
You are currently in CA
You can continue your journey to AZ NV OR
Next: ^D
clonmel$
```

Sie können Ihre Lösung wieder mit *submit* einreichen:

```
thales$ submit cpp 6 Location.?pp Territory.?pp Walk.cpp
```

Viel Erfolg!